Project 4: Scheduling and File Systems

Important Dates

Due: Thursday 04/26 11:00PM.

Overview

There are two parts to this project:

- Part A: xv6 scheduler: to be done in xv6 OS environment.
- Part B: File System:
    1. file system checker: to be done on the mumble lab, so you can learn more about programming in C on a typical UNIX-based platform (Linux).
    2. File system integrity: to be done in the xv6 environment.

Part A: xv6 scheduler

In this part, you'll be putting a new scheduler into xv6. It is called a simple **priority-based scheduler** . The basic idea is simple: assign each running process a **priority,** which is an integer number, in this case either 1 (low priority) or 2 (high priority). At any given instance, the scheduler should run processes that have the high priority (2). If there are two or more processes that have the same high priority, the scheduler should round-robin between them. A low-priority (level 1) process does NOT run as long as there are high-priority jobs available to run. Note that this is a simpler scheduler than the one discussed in the book chapter on MLFQ.

Details

You will need to implement a couple of new system calls to implement this scheduler. The first is **int setpri(int num),** which sets the priority for the calling process. By default, each process should get a priority of 1; calling this routine makes it such that a process can raise or lower its priority.

The second is **int getpinfo(struct pstat *).** This routine returns some basic information about each running process, including how long it has run at each priority (measured in clock ticks) and its process ID. You can use this system call to build a variant of the command line program **ps,** which can then be called to see what is going on.

One thing you'll have to do is make sure to initialize the priority of a process correctly. All processes, when created, should start at priority level 1. Only by calling **setpri()** can a process change its priority to 2.

When a bad priority (not 1 or 2) is passed to your setpri() system call, return -1 (indicating an error). Otherwise return 0 (indicating success). Similarly, if a bad pointer is passed to your getpinfo() call, return -1; otherwise, if the call is successful, return 0.

In your Makefile, replace `CPUS := 2` with `CPUS := 1`. The old setting runs xv6 with two CPU cores, but you only need to do scheduling for a single core in this project.

Tips

Most of the code for the scheduler is quite localized and can be found in **proc.c.** The associated header file, **proc.h** is also quite useful to examine. To change the scheduler, not much needs to be done; study its control flow and then try some small changes.

You'll need to understand how to fill in the structure **pstat** in the kernel and pass the results to user space. The structure looks like pstat.h.

Part B: File System

Part B1: File system checker

In this part, you will be developing a working **file system checker.** A checker reads in a file system image and makes sure that it is consistent. When it isn't, the checker takes steps to fix the problems it sees; however, we won't be doing any fixes this time to keep your life a little simpler.

We will use the xv6 file system image as the basic image that we will be reading and checking. The file **fs.h** includes the basic structures you need to understand, including the superblock, on disk inode format (struct dinode), and directory entry format (struct dirent). The tool **mkfs.c** will also be useful to look at, in order to see how an empty file-system image is created.

Much of this project will be puzzling out the exact on-disk format xv6 uses for its simple file system, and then writing checks to see if various parts of that structure are consistent. Thus, reading through mkfs and the file system code itself will help you understand how xv6 uses the bits in the image to record persistent information.

Your checker should read through the file system image and determine the consistency of a number of things, including the following. When one of these does not hold, print the error message (also shown below) and exit immediately.

- Each inode is either unallocated or one of the valid types (T_FILE, T_DIR, T_DEV). ERROR: **bad inode.**
- For in-use inodes, each address that is used by inode is valid (points to a valid datablock address within the image). Note: must check indirect blocks too, when they are in use. ERROR: **bad address in inode.**
- Root directory exists, and it is inode number 1. ERROR MESSAGE: **root directory does not exist.**
- Each directory contains **.** and **..** entries. ERROR: **directory not properly formatted.**
- Each **..** entry in directory refers to the proper parent inode, and parent inode points back to it. ERROR: **parent directory mismatch.**
- For in-use inodes, each address in use is also marked in use in the bitmap. ERROR: **address used by inode but marked free in bitmap.**

- For blocks marked in-use in bitmap, actually is in-use in an inode or indirect block somewhere. ERROR: **bitmap marks block in use but it is not in use.**
- For in-use inodes, any address in use is only used once. ERROR: **address used more than once.**
- For inodes marked used in inode table, must be referred to in at least one directory. ERROR: **inode marked use but not found in a directory.**
- For inode numbers referred to in a valid directory, actually marked in use in inode table. ERROR: **inode referred to in directory but marked free.**
- Reference counts (number of links) for regular files match the number of times file is referred to in directories (i.e., hard links work correctly). ERROR: **bad reference count for file.**
- No extra links allowed for directories (each directory only appears in one other directory). ERROR: **directory appears more than once in file system.**

Other Specifications

Your program must be invoked exactly as follows:

```
prompt> fscheck file_system_image
```

The image file is a file that contains the file system image. If the file system image does not exist, you should print the error **image not found.** to stderr and exit with the error code of 1. If the checker detects one of the errors listed above, it should print the proper error to stderr and exit with error code 1. Otherwise, the checker should exit with return code of 0.

Hints

It may be worth looking into using mmap() for the project.

Make sure to look at **fs.img** , which is a file system image created when you make xv6 by the tool **mkfs** (found in the tools/ directory of xv6). The output of this tool is the file **fs.img** and it is a consistent file-system image. The tests, of course, will put inconsistencies into this image, but your tool should work over a consistent image as well. Study **mkfs** and its output to begin to make progress on this project.

Part B2: File System Integrity

In this part, you'll be changing the existing xv6 file system to add protection from data corruption. In real storage systems, silent corruption of data is a major concern, and thus many techniques are usually put in place to detect (and recover) from blocks that go bad.

Specifically, you'll do three things. First, you'll modify the code to allow the user to create a new type of file that keeps a **checksum** for every block it points to. Checksums are used by modern storage systems in order to detect silent corruption.

Second, you'll have to change the file system to handle reads and writes differently for files with checksums. Specifically, when writing out such a

file, you'll have to create a checksum for every block of the file; when reading such a file, you'll have to check and make sure the block still matches the stored checksum, returning an error code (-1) if it doesn't. In this way, your file system will be able to detect corruption!

Third, for information purposes, you will also modify the `stat()` system call to dump some information about the file. Thus, you should write a little program that, given a file name, not only prints out the file's size, etc., but also some information about the file's checksums (details below).

Details

To begin, the first thing to do is to understand how the file system is laid out on disk. It is actually quite a simple layout, much like the old Unix file system we have discussed in class. As it says at the top of **fs.h** , there is a superblock, then log, some inodes, a single block in-use bitmap, and some data blocks. That's it!

What you then need to understand is what each inode looks like. Currently, it is quite a simple inode structure (found in **fs.h**), containing a file type (regular file or directory), a major/minor device type (so we know which disk to read from), a reference count (so we know how many directories the file is linked into), a size (in bytes), and then NDIRECT+1 block addresses. The first NDIRECT addresses are direct pointers to the first NDIRECT blocks of the file (if the file is that big). The last block is reserved for larger files that need an indirect block; thus, this last element of the array is the disk address of the indirect block. The indirect block itself, of course, may contain many more pointers to blocks for these larger files.

The change we suggest you make is very simple, conceptually. Your implementation should keep the inode structure exactly as is, but to use the slots for direct pointers as (checksum, pointer) pairs. Specifically, use each direct pointer slot (which are 4 bytes) as a 1-byte checksum and a 3-byte pointer. This limits how many disk addresses your file system can refer to (to 2^24), but that is probably OK for this project.

The simple one-byte checksum you will be computing over each block is XOR. Thus, to compute the checksum of a block, you should XOR all of the bytes of the block and store that value in the pointer to the block as described above. This is done every time a particular block is written, so that the file system keeps checksums up to date.

On subsequent reads of a block with a checksum, the file system should read in the block, compute its checksum, and compare the newly computed checksum to the stored checksum (the one stuffed into the pointer). If the stored and computed checksums match, your code should return as usual; if not, the call to `read()` should return an error (-1). Note that your file system does not make replicas of blocks, and thus does not have to recover from this problem; rather, you just signal an error and don't return corrupted data to the user.

Note also that large files will also have an indirect pointer allocated to them, and thus the indirect block will be full of direct pointers too, which also should have checksums in the manner described above.

One major obstacle with any file system is how to boot and test the system with the new file system; if you just change a bunch of stuff, and make a

mistake, the system won't be able to boot, as it needs to be able to read from disk in order to function (e.g., to start the shell executable).

To overcome this challenge, your file system will support two types of files: the existing pointer-based structures, and the new checksum-based structures. The way to add this is to add a new file type (see **stat.h** for the ones that are already there like T_DIR for directories, and T_FILE for files). Let's call this type T_CHECKED (a new file type, with the numeric value of 4). Thus, for regular files (T_FILE), you just use the existing code, without checksums. However, when somebody allocates an file that has extra protection (T_CHECKED), you should use your new checksum-based code.

To create an checksum-based file, you'll have to modify the interface to file creation, adding an O_CHECKED flag (value 0x400) to the `open()` system call which normally creates files. When such a flag is present, your file system should thus create an checksum-based file, with all of the expected changes as described above. Of course, various routines deeper in the file system have to be modified in order to be passed the new flag and use it accordingly.

There is no need to do any of this for directories.

Of course the real challenge is getting into the file system code and making your checksum-based modifications. To understand how it works, you should follow the paths for the `read()` and `write()` system calls. They are not too complex, and will eventually lead you to what you are looking for. Hint: at some point, you should probably be staring pretty hard at routines like **bmap()** .

Finally, you'll have to modify the `stat()` system call to return information about the actual disk addresses in the inode (stat() currently doesn't return such information). Here is the new stat structure (newstatstruct.h) you should use.

This structure has room to return exactly one byte of checksum information about the file. Thus, you should XOR all of the existing checksums over blocks of the file and return that value in the `checksum` field of the stat structure.

Also create a new program, called **filestat** , which can be called like this: **filestat pathname** . When run in such a manner, the filestat program should print out all the information about a file, including its type, size, and this new checksum value. Use this to show that your checksum-based file system works.

Submission

For the C/Linux part of this project, you should copy whatever is needed to build the checker (including a makefile) into the `linux/` subdirectory.

For the xv6 part of the project, copy all of your source files (but not .o files, please, or binaries!) into the `xv6/` subdirectory of your p4 directory. A simple way to do this is to copy everything into the destination directory directory, then type `make` to make sure it builds, and then type `make clean` to remove unneeded files.

Please submit to canvas a single archive (tar or zip) containing all xv6 files, and a short readme explaining your changes. Please name the readme *readme-username*